CSE 403
Lecture 14

Integration Testing

Reading:

The Art of Unit Testing, Ch. 1, 3, 4-5 (Osherove)
Code Complete, Ch. 29 (McConnell)

slides created by Marty Stepp
http://www.cs.washington.edu/403/

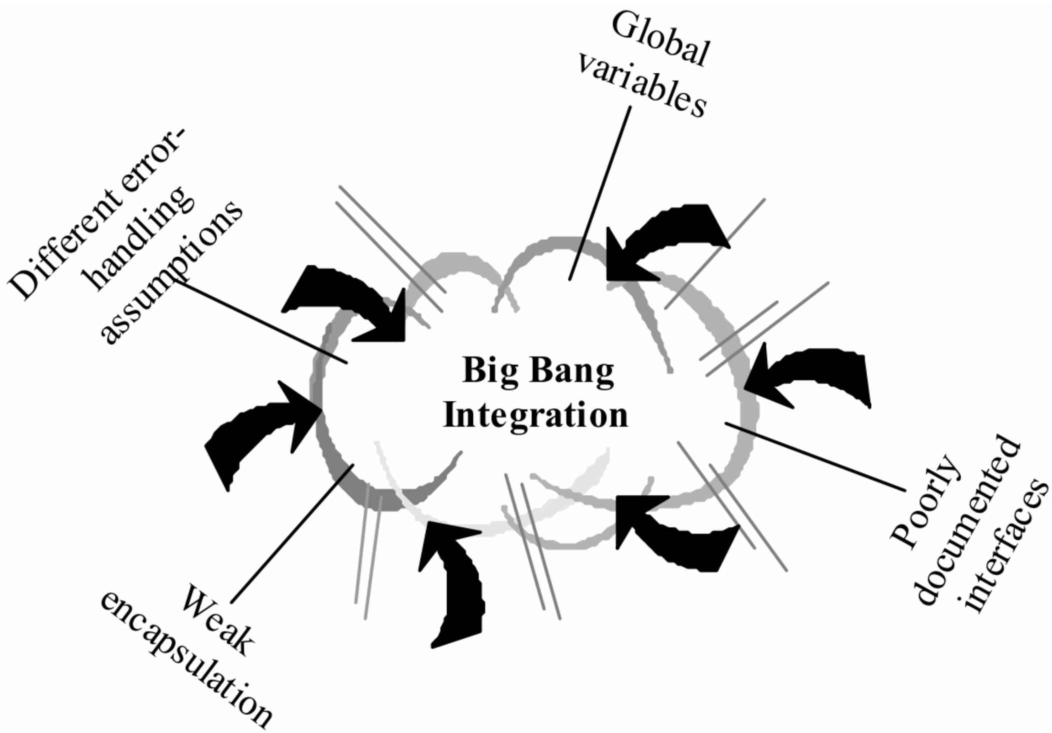
Integration

- **integration**: Combining 2 or more software units
  - often a subset of the overall project (!= system testing)

- Why do software engineers care about integration?
  - new problems will inevitably surface
    - many systems now together that have never been before
  - if done poorly, all problems present themselves at once
    - hard to diagnose, debug, fix
  - cascade of interdependencies
    - cannot find and solve problems one-at-a-time

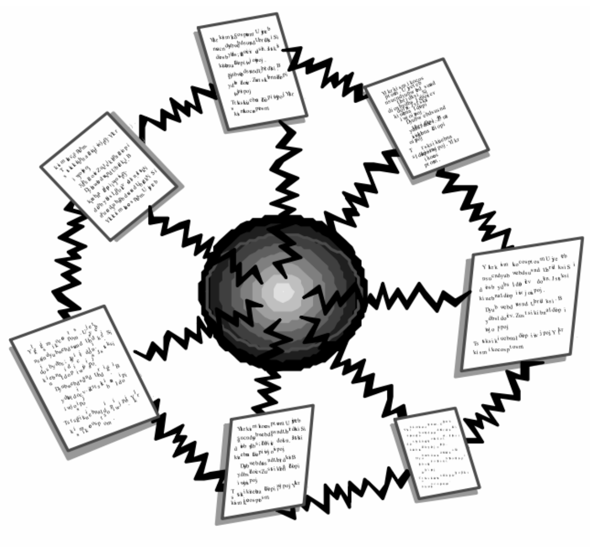
Phased integration

- **phased ("big-bang") integration**:
  - design, code, test, debug each class/unit/subsystem separately
  - combine them all
  - pray

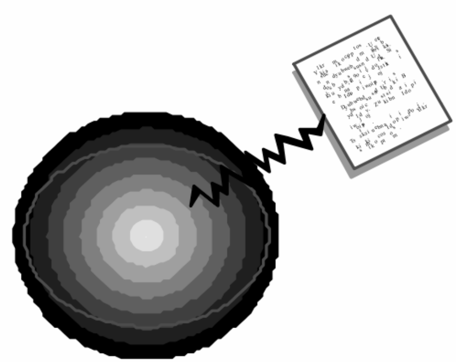


Incremental integration

- **incremental integration**:
  - develop a functional "skeleton" system (i.e. ZFR)
  - design, code, test, debug a small new piece
  - integrate this piece with the skeleton
    - test/debug it before adding any other pieces



Phased Integration



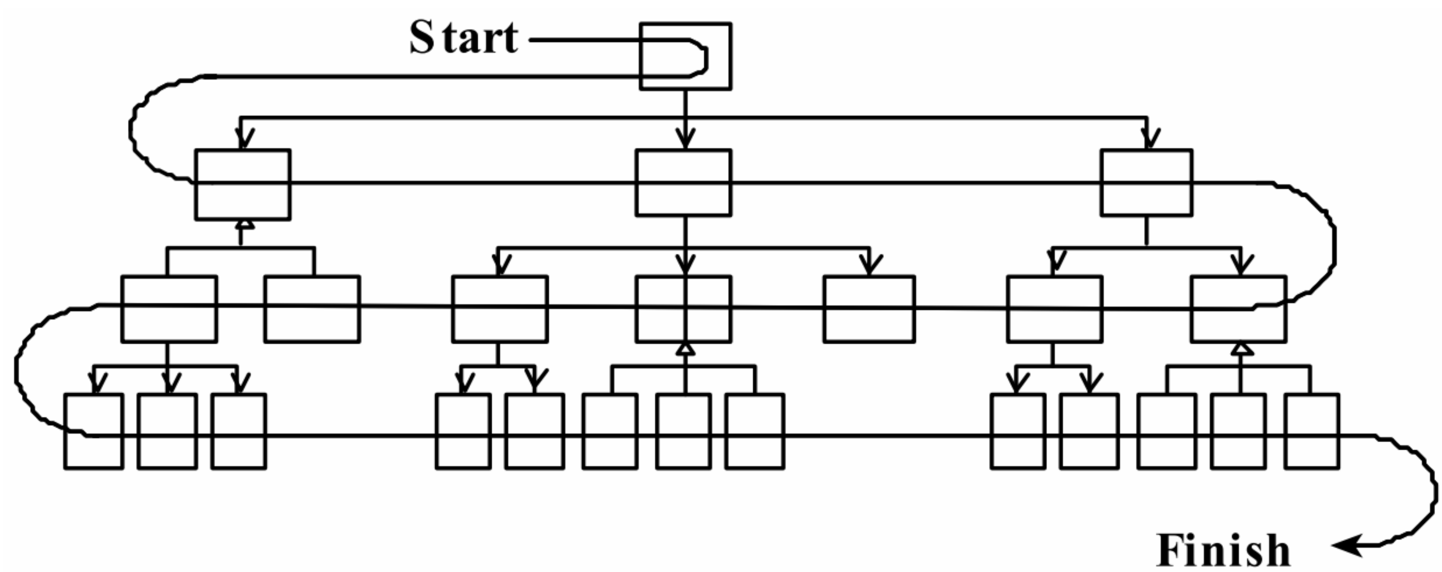
Incremental Integration

Benefits of incremental

- Benefits:
  - Errors easier to isolate, find, fix
    - reduces developer bug-fixing load
  - System is always in a (relatively) working state
    - good for customer relations, developer morale
- Drawbacks:
  - May need to create "stub" versions of some features that have not yet been integrated

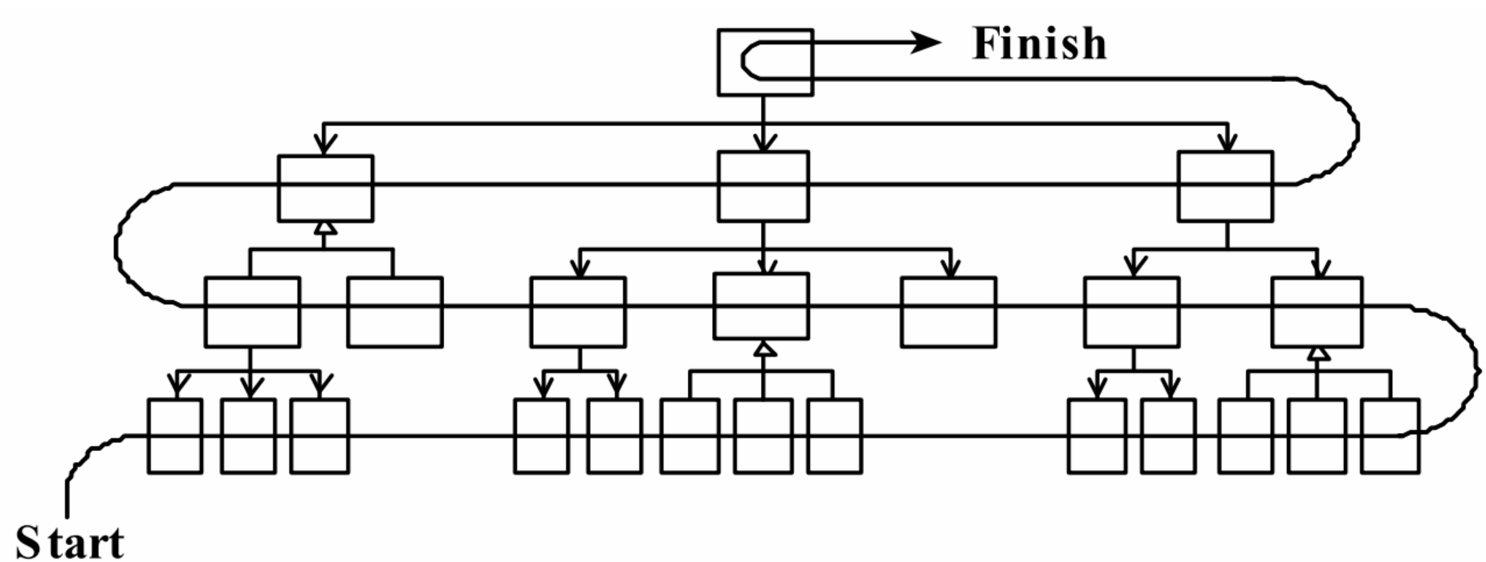
Top-down integration

- **top-down integration**:
  Start with outer UI layers and work inward
  - must write (lots of) stub lower layers for UI to interact with
  - allows postponing tough design/debugging decisions (bad?)



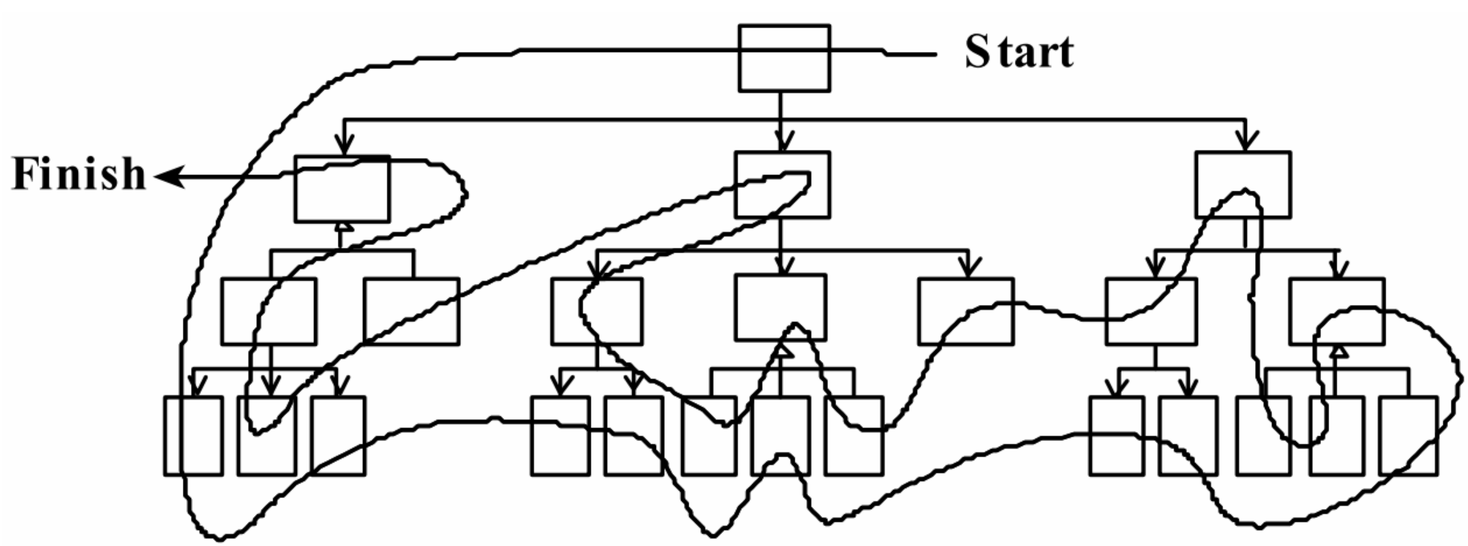
Bottom-up integration

- **bottom-up integration**:
  Start with low-level data/logic layers and work outward
  - must write test drivers to run these layers
  - won't discover high-level / UI design flaws until late



"Sandwich" integration

- **"sandwich" integration**:
  Connect top-level UI with crucial bottom-level classes
  - add middle layers later as needed
  - more practical than top-down or bottom-up?



Daily builds

- **daily build**: Compile working executable on a daily basis
  - allows you to test the quality of your integration so far
  - helps morale; product "works every day"; visible progress
  - best done *automated* or through an easy script
  - quickly catches/exposes any bug that breaks the build

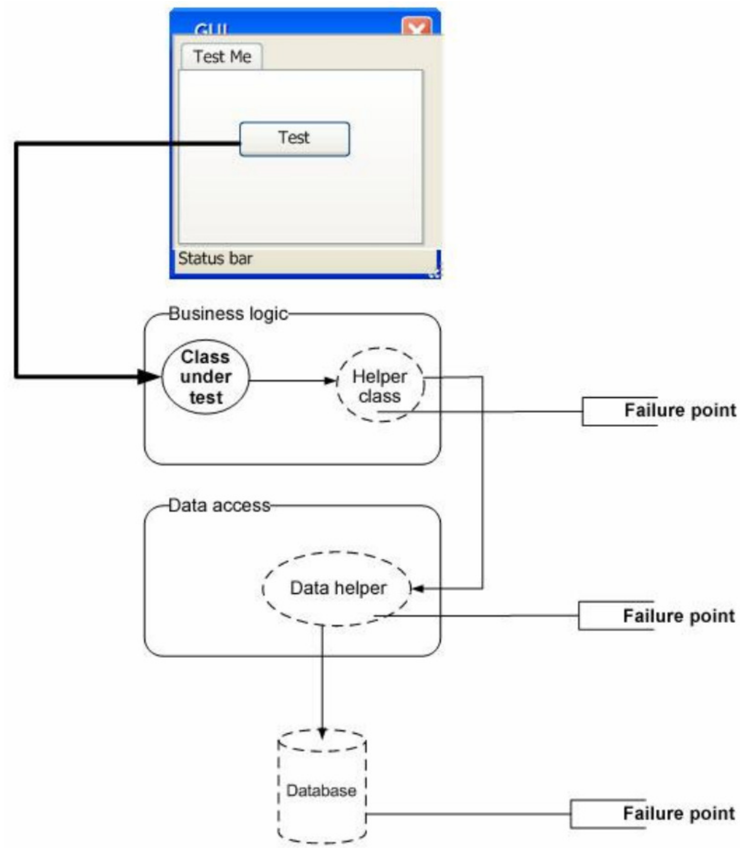
- **smoke test**: A quick set of tests run on the daily build.
  - NOT exhaustive; just sees whether code "smokes" (breaks)
  - used (along with compilation) to make sure daily build runs

- **continuous integration**:
  Adding new units immediately as they are written.

Integration testing

- **integration testing**: Verifying software quality by testing two or more dependent software modules as a group.

- challenges:
  - Combined units can fail in more places and in more complicated ways.
  - How to test a partial system where not all parts exist?
  - How to "rig" the behavior of unit A so as to produce a given behavior from unit B?



Stubs

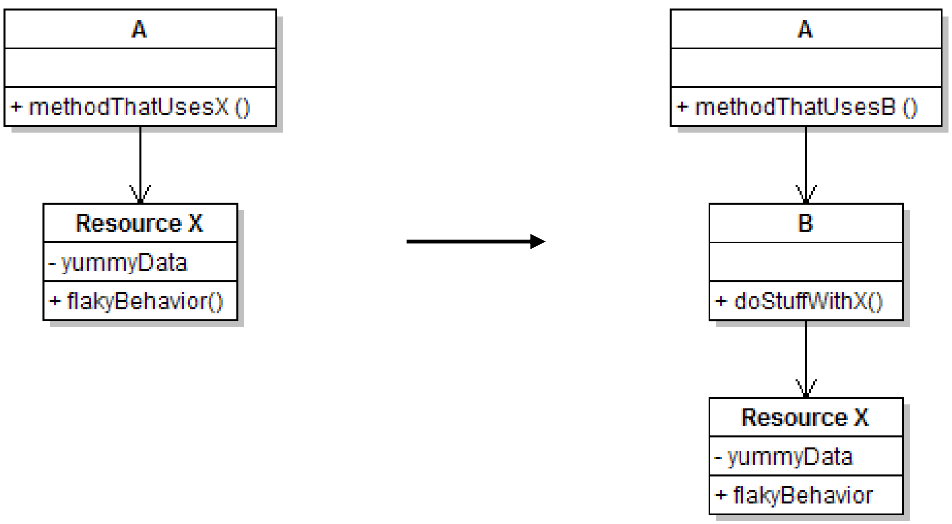
- **stub**: A controllable replacement for an existing software unit to which your code under test has a dependency.

  – useful for simulating difficult-to-control elements:
    - network / internet
    - database
    - time/date-sensitive code
    - files
    - threads
    - memory

  – also useful when dealing with brittle legacy code/systems

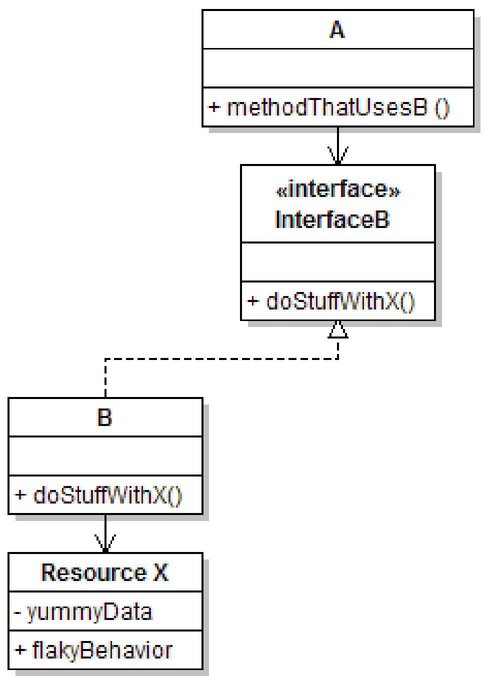
Create a stub, step 1

- Identify the external dependency.
  - This is either a resource or a class/object.
  - If it isn't an object, wrap it up into one.
    - (Suppose that Class A depends on troublesome Class B.)



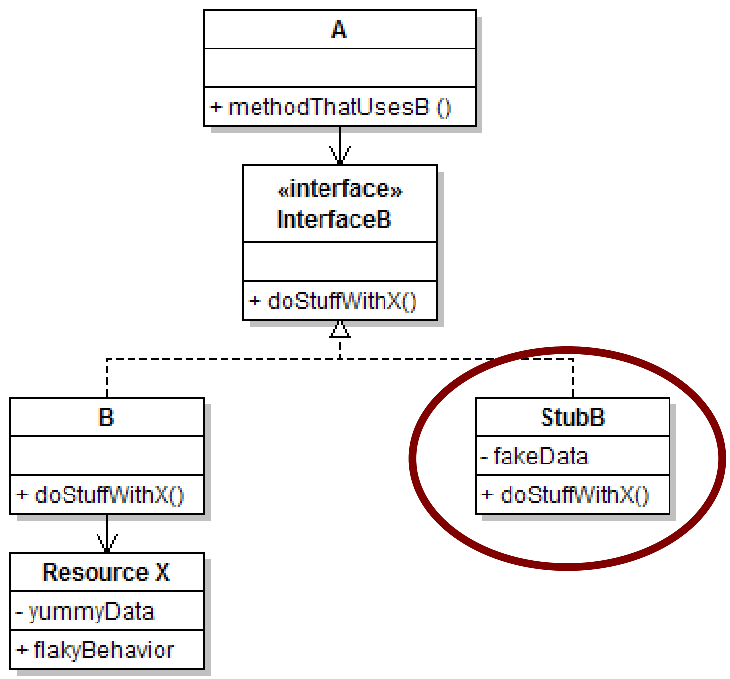
Create a stub, step 2

- Extract the core functionality of the object into an interface.
  - Create an InterfaceB based on B
  - Change all of A's code to work with type InterfaceB, not B



Create a stub, step 3

- Write a second "stub" class that also implements the interface, but returns pre-determined fake data.
  - Now A's dependency on B is dodged and can be tested easily.
  - Can focus on how well A *integrates* with B's external behavior.



Injecting a stub

- **seams**: Places to inject the stub so Class A will talk to it.
  - at construction (not ideal)

    ```
    A aardvark = new A(new StubB());
    ```

  - through a getter/setter method (better)

    ```
    A apple = new A(...);
    aardvark.setResource(new StubB());
    ```

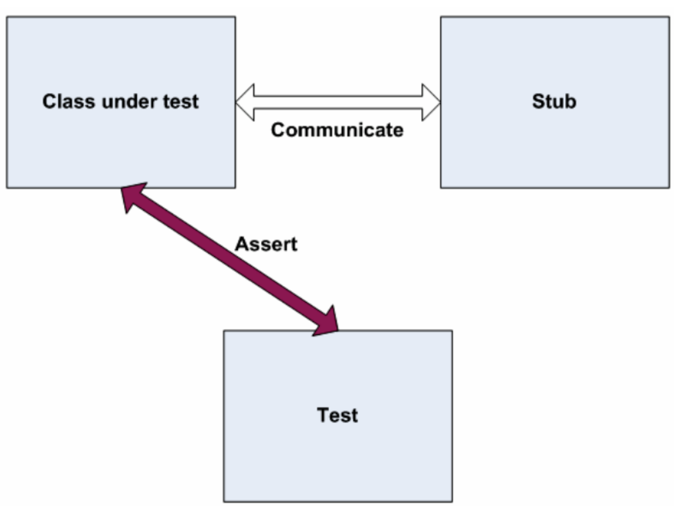
  - just before usage, as a parameter (also better)

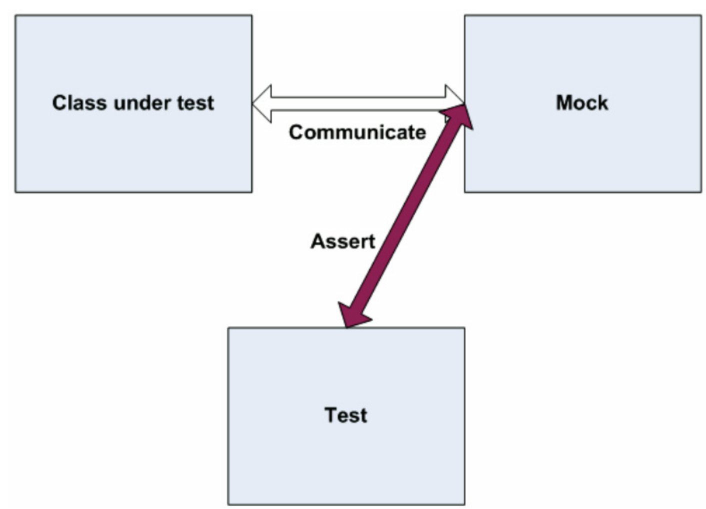
    ```
    aardvark.methodThatUsesB(new StubB());
    ```

    - You should not have to change A's code everywhere (beyond using your interface) in order to use your Stub B. (a "testable design")

"Mock" objects

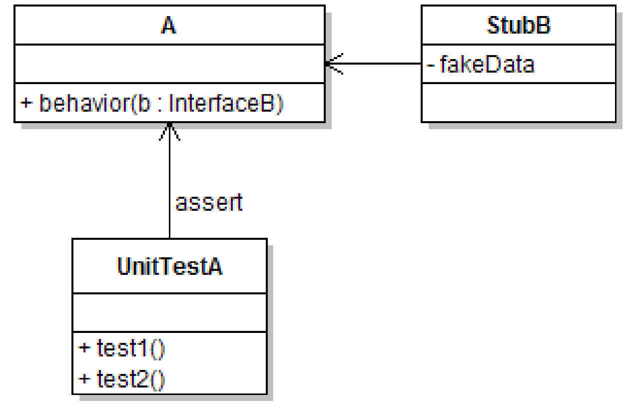
- **mock object**: A fake object that decides whether a unit test has passed or failed by watching interactions between objects.
  - useful for **interaction testing** (as opposed to **state testing**)



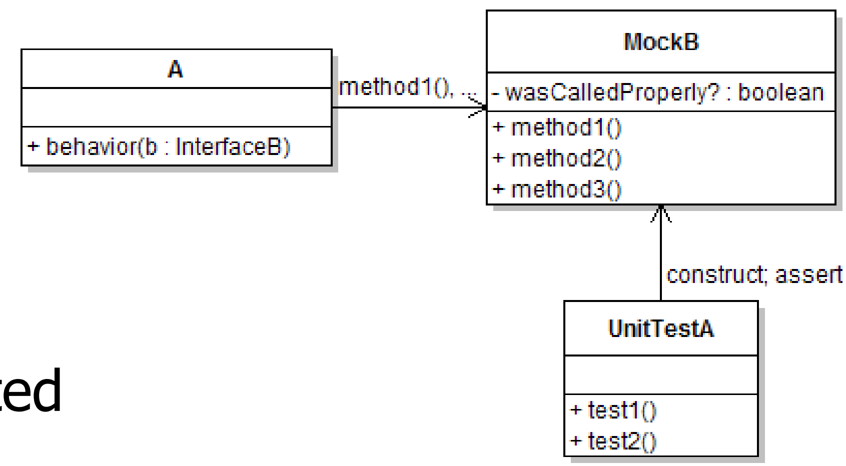


Stubs vs. mocks

- A **stub** gives out data that goes to the object/class under test.
- The unit test directly asserts against class under test, to make sure it gives the right result when fed this data.



- A **mock** waits to be called by the class under test (A).
  - Maybe it has several methods it expects that A should call.
- It makes sure that it was contacted in exactly the right way.
  - If A interacts with B the way it should, the test passes.



Mock object frameworks

- Stubs are often best created by hand/IDE.
  Mocks are tedious to create manually.

- Mock object frameworks help with the process.
  - android-mock, EasyMock, jMock (Java)
  - FlexMock / Mocha (Ruby)
  - SimpleTest / PHPUnit (PHP)
  - ...

- Frameworks provide the following:
  - auto-generation of mock objects that implement a given interface
  - logging of what calls are performed on the mock objects
  - methods/primitives for declaring and asserting your expectations

A jMock mock object

```
import org.jmock.integration.junit4.*;  // Assumes that we are testing
import org.jmock.*;                     // class A's calls on B.

@RunWith(JMock.class)
public class ClassATest {
    private Mockery mockery = new JUnit4Mockery();  // initialize jMock

    @Test  public void testACallsBProperly1() {
        // create mock object to mock InterfaceB
        final InterfaceB mockB = mockery.mock(InterfaceB.class);

        // construct object from class under test;  attach to mock
        A aardvark = new A(...);
        aardvark.setResource(mockB);

        // declare expectations for how mock should be used
        mockery.checking(new Expectations() {{
            oneOf(mockB).method1("an expected parameter");
            will(returnValue(0.0));
            oneOf(mockB).method2();
        }});

        // execute code A under test; should lead to calls on mockB
        aardvark.methodThatUsesB();

        // assert that A behaved as expected
        mockery.assertIsSatisfied();
    }
}
```

jMock API

- jMock has a strange API based on "Hamcrest" testing syntax.

- Specifying objects and calls:
  - `oneOf(`**mock**`)`, `exactly(`**count**`).of(`**mock**`)`,
  - `atLeast(`**count**`).of(`**mock**`)`, `atMost(`**count**`).of(`**mock**`)`,
  - `between(`**min**`,` **max**`).of(`**mock**`)`
  - `allowing(`**mock**`)`, `never(`**mock**`)`

    - The above accept a mock object and return a descriptor that you can call methods on, as a way of saying that you demand that those methods be called by the class under test.

  - `atLeast(3).of(mockB).method1();`
    - "I expect that `method1` will be called on `mockB` 3 times here."

Expected actions

- `.will(action)`
  - actions: `returnValue(`**v**`)`, `throwException(`**e**`)`

- values:
  - `equal(`**value**`)`, `same(`**value**`)`, `any(`**type**`)`, `aNull(`**type**`)`, `aNonNull(`**type**`)`, `not(`**value**`)`, `anyOf(`**value1, ..,valueN**`)`

  - `oneOf(mockB).method1();`
    `will(returnValue(anyOf(1, 4, -3)));`
    - "I expect that `method1` will be called on `mockB` once here, and that it will return either 1, 4, or -3."

Using stubs/mocks together

- Suppose a log analyzer reads from a web service.
  If the web fails to log an error, the analyzer must send email.
  - How to test to ensure that this behavior is occurring?

- Set up a *stub* for the web service that intentionally fails.
- Set up a *mock* for the email service that checks to see whether the analyzer contacts it to send an email message.